John von Neumann Institute for Computing

JPEG2000 Optimization in General Purpose Microprocessors

C. García, C. Tenllado, L. Piñuel, M. Prieto

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 599-606, 2006.

http://www.fz-juelich.de/nic-series/volume33

JPEG2000 Optimization in General Purpose Microprocessors*

C. García[a], C. Tenllado[a], L. Piñuel[a], M. Prieto[a]

[a]Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense, 28040 Madrid, Spain
e-mail: {garsanca, tenllado, lpinuel, mpmatias}@dacya.ucm.es

We have addressed in this paper the efficient implementation of the JPEG2000 in high-end microprocessor. From a computational perspective, the demands of this new standard, which is substantially more complex than its predecessor, makes this goal a challenging but extremely important task. Previous work about this topic has focused on data locality issues. Our efforts are aimed at improving the exploitation of *Multimedia ISA Extensions* and *Simultaneous Multithreading*. Performance results obtained on an Intel Pentium 4 processor show impressive speedups, that range from 1.9 to 22 depending on the target image size.

1. Introduction

Increasing focus on multimedia applications has prompted many researchers to design efficient image and video coding systems. An outstanding example of this effort is the JPEG2000, the latest series of standards from the JPEG committee. Apart from achieving higher compression rates than its predecessor (popularly referred to as JPEG), it is capable of handling many more aspects than simply making the digital image files as small as possible. However, from a computational perspective it is substantially more complex than JPEG, which makes its implementation a challenging task.

In the computer architecture area, this focus has also prompted the adaptation of general purpose architectures for multimedia workloads. Basically, functional units have been enhanced with *Subword Level Parallelism* (SLP) capabilities and the instruction-set architectures (ISA) have been extended to include new instructions (*Multimedia ISA Extensions*) that operate on packed data. Well-known examples are the Intel's SSE/SSE2/SSE3 ISA extensions of the IA-32 family [23] and the IBM-Motorola's Altivec [7]. However, the use of such extensions is difficult. Compiler technology is still not highly developed in this field and programmers are usually restricted to using in-line assembly, intrinsic functions or specialized libraries [18].

Aside from these extensions, multimedia codes can also take advantage of the additional levels of parallelism available in high-end microprocessors, such as *Thread-Level Parallelism* (TLP). *Single-Chip Multiprocessors* (CMP) are expected to soon be ubiquitous [3] and most superscalar-style cores likely will have some form of *Simultaneous multithreading* (SMT). In particular, SMT has already been incorporated into the Intel's Pentium 4 and Xeon families [14] and into the IBM's Power5 [11].

Our main goal in this paper is to study how to adapt the JPEG2000 explicitly to take advantage of both SLP and TLP parallelism on SMT architectures. The limitations of current compiler technology and the importance of this standard, makes this study of great practical interest. In addition, it also provides certain insights about the potential benefits of these relatively new capabilities and how to take advantage of them, which can help to develop more efficient compilers.

Our target code is a reference implementations of the standard, known as *JasPer* [1], which implements the codec specified in the JPEG2000 Part-1 standard (i.e., ISO/IEC 15444-1). For the

*This work has been supported by the Spanish government research contract TIC 2002-750 and the Hipeac European Network of Excellence

sake of conciseness, we have focused on the lossy compression process. Nevertheless, the proposed methodology can also be applied to both the lossless mode and the decoding process. Experiments have been performed on an Intel Pentium 4 running at 3.4 Ghz (2MB L2 cache, 1Gb DDR400).

The rest of the paper is organized as follows: some related work is summarized in Section 2; Sections 3 and 4 describe the proposed optimizations and present some performance results. Finally the paper ends with some conclusions.

2. Related Work

A significant amount of work on the optimization of the *lifting*-based 2D discrete wavelet transform (DWT) has been performed in recent years within the JPEG2000. This interest is generated by the considerable percentage of execution time involved in this component of the standard (around 40-60% according to some authors [19]). Most optimizations have been focused on improving cache reuse [15,16,4]. In [15], authors addressed the optimization of *JasPer* and *jj2000* (another reference implementation of the JPEG2000 written in java) by means of traditional *loop-tiling* and *array-padding* techniques (denoted by them as *aggregation* and *row-extensions*). Despite the relative simplicity of the proposed optimizations, the combined effect of both techniques on the DWT vertical filtering (the one that lacks spatial locality if the image is stored following a row-major layout) achieves a speed-up factor of around 10 for large image sizes. This tremendous improvement translates into an overall coding time reduction of around 2. This research was extended in [16] to shared-memory symmetric multiprocessors, applying the general principles of data-domain decomposition. Apart from *loop-tiling* techniques, [4] investigates the use of specific array layouts as an additional means of improving data locality.

Regarding SLP, we should mention [21], where a fixed-point implementation of the DWT has been vectorized using Intel's MMX. Related work centered on multithreaded architectures is also scarce. We can mention [12], in which a load adaptive approach for fine-grain multithreading architectures is proposed.

3. Exploiting SLP in the JPEG2000

Conventional vectorization techniques were designed during the 70's and the 80's to extract parallelism from computational intensive Fortran programs. Today, these techniques are being adapted to support the short-vector processing capabilities found in modern microprocessors and to take into account the requirements of multimedia workloads. In the commercial marketplace, we should highlight for example the continuous efforts being made by Intel, IBM and the Portland Group in their respective compiler infrastructures [2,9,8]. Within the academic community, we should mention the research of Larsen et al. [13] and the new release of the open-source compiler GCC 4.1, which has been significantly enhanced to support automatic vectorization [17].

Despite these extensive efforts, state-of-the art compilers cannot automatically vectorize any component of the *JasPer* library from the scratch.

3.1. Code Analysis

The compiler limitations mentioned above forced us to conduct an in-depth code analysis to find out which procedures are susceptible of SLP exploitation. The analysis was guided by the hints provided by the Intel C/C++ Compiler v.8.1 [10], which help us to identify vectorization inhibitors, and by an extensive profiling that helped us to select the most demanding components. Instead of using the original *JasPer*, our baseline code is a hand-tuned version, which already includes a

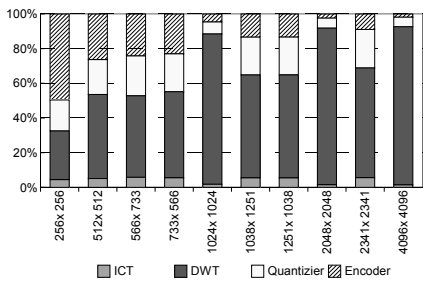

Figure 1. Execution time breakdown of the Original *JasPer's* lossy compression process for different images sizes.

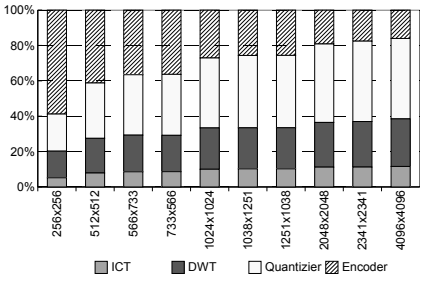

Figure 2. Execution time breakdown of our cache-aware implementation of the *JasPer's* lossy compression process for different images sizes.

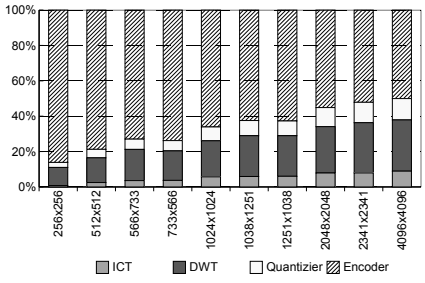

Figure 3. Execution time breakdown of our cache-aware implementation of *JasPer's* lossy compression process turning on SLP optimizations.

cache-aware DWT implementation (more details in Section 3.3).

As a result of this analysis, we identified three potential candidates in the lossy compression process: the *Irreversible Color Transform* (ICT), the *Discrete Wavelet Transform* (DWT) and the *Quantization* process.

The *encoding/decoding* stages have not been considered since they are characterized by substantial data and control dependencies, as well as irregular memory access patterns, limiting opportunities for fine-grain data parallelism exploitation.

Figures 1 and 2 show execution time breakdowns of the original and our baseline *JasPer's* implementation respectively. The reported percentages correspond to the coding of the *Lena* color image (treated as a single tile), using lossy compression with full bit-rate. The image sizes have been chosen following the results reported in [4]. Each bar shows the relative execution cost of the most important components of this process (excluding I/O operations).

It is worth to note, that despite the impressive enhancements achieved by the memory optimizations introduced in the DWT (discussed in Subsection 3.5), the execution time is still dominated by the three vectorizable stages, leaving room for significant improvements in performance through SLP exploitation. In the following subsections we describe in detail how these three components were vectorized.

3.2. Irreversible Color Transform (ICT)

In the lossy compression process a classic forward inter-component transformation maps the data from the RGB to the YCbCr color space. It operates on all of the components together, and serves to reduce the correlation between components. The corresponding *for* loop nest that implements this linear transformation can be easily vectorized since it does not present dependencies and data items are sequential in memory. Spatial locality cannot be further improved and its nature make optimizations aimed at improving temporal locality ineffective, as no data reuse exist. However, this transformation was originally coded in *JasPer* using its own fixed-point arithmetic template class (a set of macros), which includes certain datatype conversions (*castings*) that inhibit automatic vectorization. On the other hand, some difficulties also arise in the vectorization of fixed-point computations. For instance, integer multiplication involves intermediate data twice as wide as the original operators, which prevents full SIMD exploitation. These difficulties lead us to perform a floating-point ICT transformation that produces exact agreement (within round-off error).

3.3. Discrete Wavelet Transform (DWT)

The algorithm used to compute the DWT in the JPEG2000 standard is the *Lifting* scheme [20].

Despite the inherent data parallelism of the DWT, *JasPer's* fixed-point arithmetic class, RAW dependencies between the different lifting steps, and strided memory accesses inhibit the automatic vectorization of this important component.

3.3.1. Vectorization of the Vertical filtering

Guided in part by our previous work [5], we have introduced the following optimizations into the *JasPer's* implementation of the DWT:

- *Loop fusion*: The *prediction*, *update* and *normalization* steps of the lifting [20] scheme have been fused in only one loop, enhancing temporal locality.

- *Loop interchange*. The spatial locality of the vertical filtering has been significantly improved using a loop-interchange transformation. The naïve implementation used in *JasPer* for this filtering suffers from an important memory bottleneck, given that data is processed by columns despite using a row-major layout. As a positive side-effect, this loop transformation moves the RAW dependencies to the outer loop, enabling the vectorization of this filtering [5].

- *Pipeline computation and array padding*. Array memory access has been further improved using *pipeline computation* [5], which enhances temporal locality, as well as using *array padding*, which reduces data cache conflict misses.

These optimizations, apart from improving memory hierarchy exploitation, allow for an efficient and straightforward vectorization of the vertical filtering performing the lifting steps of four consecutive columns in parallel. Among the different memory management alternatives we have employed the *inplace-mallat* strategy that we introduced in [6]. For the sake of conciseness, we refer to this previous paper for a more elaborate description.

3.3.2. Vectorization of the Horizontal filtering

Although *loop fusion*, *pipeline computation* and *array padding* also improve the performance of the horizontal filtering, its vectorization is a more challenging task. If data is processed row-by-row, which maximizes spatial locality, the inner loop of the horizontal filtering present RAW data dependencies that prevent vectorization. In contrast, if a loop interchange transformation is applied to move RAW dependencies out of the inner loop, strided memory access are necessary, degrading performance. Commercial compilers do not apply such transformations since their heuristics suggest that the vectorization benefits does not compensate the overheads caused by strided memory access pattern. A trade-off between memory access and inner-loop parallelism is necessary.

Transposing the whole array to allow for the same strategy employed in the vertical filtering is not feasible since the overhead of the transposition is larger than the benefits of vectorization. However, in [5] we found out that the combination of *pipeline computation* and a non-linear data layout, makes local transposition of small array tiles affordable. Unfortunately, we have ruled out this approach given that the integration of a non-linear layout within the JPEG2000, not only involves a significant coding effort, but also important overheads due to memory transfers in other stages of the coding process. Alternatively, we have adapted the methodology presented in [5] to a row-major data layout. The adaptation has been performed introducing the following optimizations in the horizontal filtering:

- The outer loop has been tiled so that several rows are processed simultaneously.

- The vector register file is used as a temporal buffer to hold four vectors loaded from consecutive rows. The vector block is then efficient transposed without needing additional memory accesses. To further improve memory access, the inner loop has been unrolled so that all vectorial loads are memory aligned.

- Temporal locality is further optimized storing the output vectors back to memory when they are no longer needed in the horizontal filtering, i.e. we perform a manual scalar replacement on vector data.

In summary, the core of our new strategy to vectorize the horizontal filtering is a local transposition that at the expense of additional data movements allows for performing the lifting steps of four consecutive rows in parallel, i.e. it enables the same strategy applied on the vertical filtering. Nevertheless, it is worth to note that without the memory optimization mentioned above, this transposition is quite inefficient.

3.4. Quantization

After transforming the image components, the real wavelet coefficients are quantized to an integer space. As in the ICT stage, the memory accesses in the corresponding loop nest cannot be further improved, but the vectorization is feasible since no data dependencies exist and data items are sequential in memory. However, the employment of the *JasPer's* fixed-point arithmetic class inhibit vectorization and we opted to perform intermediate computations using floating-point. In addition, we have removed the *if-statement* included in the original's quantization inner loop adding some extra arithmetic. This conditional statement, used to distinguish between positive and negative data, introduces unnecessary control dependencies that hamper performance. Our transformation not only makes vectorization possible but also enhances dramatically the performance of this stage, which already runs around 5 times faster than the original *JasPer's* quantization even without enabling vectorization.

3.5. Performance Results

We have reported separately in Tables 1 and 2 the speedups achieved by the memory-hierarchy and SLP optimizations. To isolate the different contributions to the overall speedup, these figures do not take into account the additional gains introduced by substituting *JasPer's* fixed-point arithmetic class and by removing *if-statements* in the quantization stage.

Table 1
Speedups achieved by the proposed memory optimizations. Local and Global denote the speedups on the DWT and how them translate into the whole compression process respectively.

Image Size	Local	Global
256x256	2.19	1.26
512x512	3.82	1.63
566x733	3.39	1.52
733x566	3.80	1.60
1024x1024	21.24	5.48
1038x1251	4.82	1.81
1251x1038	4.80	1.80
2048x2048	27.34	6.71
2341x2341	4.98	1.82
4096x4096	28.08	7.10

Table 2
Speedups achieved by SLP exploitation. ICT, DWT and QT stand for the local speedup on these stages while Global refers to the full compression process.

Image Size	ICT	DWT	QT	Global
256x256	3.51	1.66	3.97	2.95
512x512	2.31	1.95	3.64	2.45
566x733	1.84	1.72	3.43	2.17
733x566	1.83	1.84	3.52	2.20
1024x1024	1.76	2.08	3.70	1.96
1251x1038	1.66	1.84	3.44	1.82
1038x1251	1.67	1.89	3.66	1.85
2048x2048	1.76	2.20	3.68	1.75
2341x2341	1.65	1.96	3.38	1.64
4096x4096	1.77	2.26	3.72	1.72

In the DWT, the gains achieved through memory optimization increase with the image size as could be expected, although power of two image sizes represent a pathological case due to the impact of conflict cache misses, which are almost completely removed by means of array padding. The proposed vectorization delivers consistent speedups that range between 1.5 and 2.3 in the horizontal filtering and between 1.5 and 3.9 in the vertical one, which translates into an average overall speedup close to 2. The speedups achieved on the ICT and the Quatization stages are also consistent and very close to the ideal values.

Finally, Figure 3 shows the new execution time breakdown. The encoding stage becomes now the most demanding procedure and hence, further optimization efforts should be focused, directly or indirectly, on this stage.

4. Simultaneous Multithreading

In a previous work [22] we proposed an alternative approach to compute the DWT on SMT architectures based on a functional partitioning strategy. It showed to be more efficient than traditional data partitioning techniques, frequently used in shared memory multiprocessors, achieving an extra 30% of speedup. However, given that the DWT stage only accounts for around 15%-25% of the JPEG compression process after memory and SLP optimizations (see Figure 3), this strategy does not promise significant returns. Nevertheless, based on that approach, our intuition was that a functional partitioning of the whole compression process would likely be efficient. For color images, where different channels have to be processed, this partitioning can be performed using a pipeline strategy.

Given that our SMT architecture only allows for two simultaneous threads, we have split the lossy compression process into two stages. Taking into account load balancing issues, the first stage performs DWT and Quantization, whereas the second stage performs the encoding (tier1 and tier2).

Figure 4 shows the benefits of the proposed parallelization strategy on the target SMT platform.We have considered two parallel versions, with and without enabling SLP. The speedups are quite satisfactory (higher than 15% in most cases) taking into account both, the expected gains reported by Intel (around 30%) [14], and the parallel fraction of code ($2/3$). It is also worth to note that the performance improvements are higher when SLP is enable, which reproduces the behavior observed for the DWT in [22]. Figure 5 analyzes this synergy between SLP and SMT. The stripped bars show the overall speedup that could be expected by applying both SLP and SMT optimizations if both improvements were independent (i.e. assuming a multiplicative effect), whereas the black bars show the actual speedups. The latter speedups are always higher for all the image sizes.

Finally, Table 4 shows the impressive overall speedups achieved by our proposed implementation over the original *JasPer*. The overall gains range between 1.9 and 22 times depending on the image size.

5. Conclusions

The JPEG2000 is a well known image coding standard which is used nowadays in many multimedia applications and whose popularity will surely grow in the next few years. Its tuning for different platforms is becoming crucial for many imaging applications. Some promising sources for optimization are the exploitation of SLP and SMT.

Vectorization has been a hot topic since the introduction of vector supercomputers. However, when classical techniques are applied to SLP code generation, they can introduce a considerable overhead in the resulting codes. In this paper we have applied and extended a systematic approach

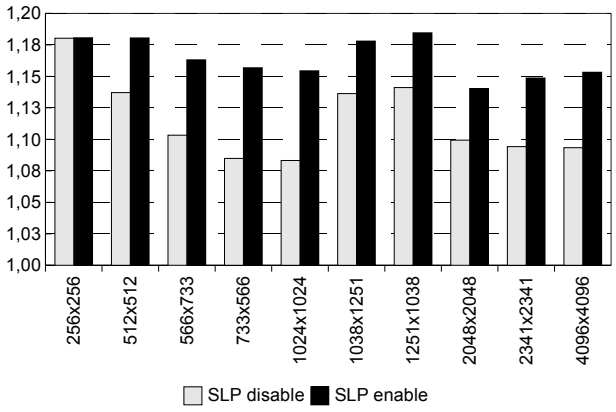

Figure 4. Speedup of the parallel version with and without turning on SLP.

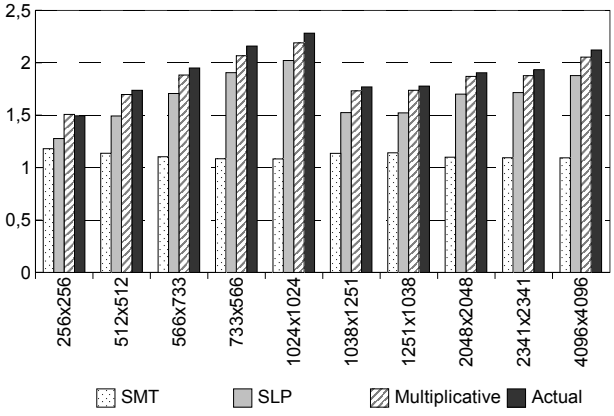

Figure 5. Synergy between SLT and SMT.

Table 3
Execution times (in milliseconds) of the JPEG2000 compression process. The column labeled as *Reference* refers to the original *JasPer*, whereas *Tuned* refers to our proposed implementation with all the optimizations turned on.

Image size	Reference(ms)	Tuned (ms)	SpeedUp
256x256	52	27	1,9
512x512	180	57	3,1
566x733	254	81	3,1
733x566	264	80	3,3
1024x1024	1993	157	12,6
1038x1251	836	192	4,3
1251x1038	828	188	4,4
2048x2048	9473	491	19,2
2341x2341	3820	642	5,3
4096x4096	39711	1780	22,3

applied in [22] to the DWT. Although hand-tuning has been needed to obtain satisfactory results, some general rules have been established to achieve an efficient implementation. Three stages in the lossy compression process have been successfully vectorized and our previous research about DWT vectorization has been extended with a new vectorization of the horizontal filtering based on a local transposition.

Furthermore, the compression process has been multi-threaded taking advantage of the functional parallelism available in multi-component images.

Overall, the performance of the *JasPer* implementation has been improved by a factor that ranges from 1.9 to 22, depending on the image size. In a future research we plan to apply this methodology to Motion JPEG2000, in an attempt to meet real-time video coding requirements.

References

[1] M. Adams and R. Ward. Jasper: A portable flexible open-source software tool kit for image coding/processing. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 2004.

[2] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. of Parallel Programming*, 30(2):65–98, 2002.

[3] Doug Burger and James R. Goodman. Billion-transistor architectures: There and back again. *Computer*,

37(3):22–28, Mar. 2004.

[4] S. Chatterjee and C. D. Brooks. Cache-effi cient wavelet lifting in JPEG 2000. In *Proc. of the IEEE Int. Conf. on Multimedia and Expo*, pages 797–800, Lousanne, Switzerland, Aug. 2002.

[5] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation. In *Proc. of the 2002 Int. Conf. on High Performance Computing (HiPC'02)*, pages 9–21, India, Dec. 2002.

[6] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. Vectorization of the 2d wavelet lifting transform using SIMD extensions. In *Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia (PDIVM'03)*, Nize, France, Apr. 2003.

[7] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[8] The Portland Group. Pgi users guide : Parallel Fortran, C and C++ for scientists and engineers. Information available at http://www.pgroup.com/doc/pgiug.pdf.

[9] IBM Corporation. XL C/C++ advanced edition for linux. Information available at http://www-306.ibm.com/software/awdtools/xlcpp/features/linux/index.html.

[10] Intel. Intel C/C++ compilers for Linux. http://www.intel.com/software/products/compilers.

[11] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. 24(2):40–47, 2004.

[12] A. Khokhar, G. Hebe, P. Thulasiraman, and G. R. Gao. Load adaptive algorithms and implementations for the 2d discrete wavelet transform on fi ne-grain multithreaded architectures. In *Proc. of the Int. Parallel Processing Symp. (IPPS/SPDP 1999)*, Puerto Rico, Apr. 1999.

[13] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming language design and implementation (PLDI '00)*, pages 145–156, New York, NY, USA, 2000. ACM Press.

[14] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology J.*, 6(1):4–15, Feb. 2002.

[15] P. Meerwald, R. Norcen, and A. Uhl. Cache issues with JPEG2000 wavelet lifting. In *Proc. of 2002 Visual Communications and Image Processing (VCIP'02)*, pages 626–634, 2002.

[16] P. Meerwald, R. Norcen, and A. Uhl. Parallel JPEG2000 image coding on multiprocessors. In *Proc. of the Int. Parallel and Distributed Processing Symp.(IPDPS'02)*, Florida, Apr. 2002.

[17] Dorit Naishlos. Autovectorization in gcc. In *Proc. of the GCC Developers Summit*, pages 105–118, Ottawa, Canada, Jun. 2004.

[18] Gang Ren, Peng Wu, and David Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS05)*, Denver, USA, Apr. 2005.

[19] D. Santa-Cruz and T. Ebrahimi. A study of JPEG 2000 still image coding versus other standards. In *Proc. of the X European Signal Processing Conf.*, volume 2, pages 673–676, Finland, Sep. 2000.

[20] Wim Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. on Mathematical Analysis*, 29(2):511–546, 1998.

[21] S. Taubman and M. W. Marcellin. *JPEG2000: Image Compression Fundamentals, Standards, and Practice*. Kluwer Int. Series in Engineering and Computer Science, 2002.

[22] C. Tenllado, C. Garc´ıa, M. Prieto, L. Piñuel, and F. Tirado. Exploiting multilevel parallelism within modern microprocessors: DWT as a case study. In *Post-Conf. book of Vecpar'04*, pages 556–568, 2004.

[23] Shreekant (Ticky) Thakkar and Tom Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.